
# Proceedings of the Third International Workshop on Sustainable Ultrascale Computing Systems (NESUS 2016) Sofia, Bulgaria

Jesus Carretero, Javier Garcia Blas, Svetozar Margenov
(Editors)

October, 6-7, 2016

Burrows, E., Friis, H. A. & Haveraaen, M. (2016). An Array API for FDM. En *Proceedings of the Third International Workshop on Sustainable Ultrascale Computing Systems (NESUS 2016) Sofia, Bulgaria* (pp. 59-68). Madrid: Universidad Carlos III de Madrid. Computer Architecture, Communications, and Systems Group (ARCOS).

# An Array API for FDM

EVA BURROWS† HELMER ANDRÉ FRIIS* MAGNE HAVERAAEN†‡

† Bergen Language Design Laboratory
Department of Informatics, University of Bergen, Norway
https://bldl.ii.uib.no/

* IRIS
Stavanger, Norway
http://www.iris.no/

**Abstract**

*As we move towards ultrascale computing, computer architecture is bound to see dramatic changes. Multiple nodes, with or without shared memory, multicore and accelerators (GPUs, FPGAs) will be the norm. For many problems, such as finite difference numerical simulations, the array used to represent a perfect match between the user level code and the hardware architecture's uniform memory access. Arrays, and to some extent multiarrays, are well supported by most programming languages. A standard compiler maps the array for uniform memory. Some programming models, such as partitioned global address space, allows mapping an array across distributed, yet for each partition, uniform memory. For ultrascale architectures, the simple mapping between user level (multi)array and distributed, non-uniform memory, will disappear. Here we propose an API for arrays, empowering the software developer to implement their own array-memory layout. Application code written towards the API will be independent of underlying architecture changes, thus easily ported to new architectures as they evolve.*

***Keywords*** Scientific Computing, Ultrascale Computing, Multiarray API, Array API for Finite Difference Methods

## I. INTRODUCTION

The array has been a central concept for software development, especially in the high performance domain. For instance, multiarrays are key to explicit finite difference solvers for partial differential equations (PDE). Languages for programming in computational science have direct language support for arrays. Many also directly support multidimensional array manipulations (e.g. MATLAB, Fortran, F), or introduce libraries or packages to support this (e.g. Boost.MultiArray Library for C++, NumPy for Python). In the problem space, the array provides an abstraction for indexable data collections. In the hardware space, the array represents linear addressable memory.

Compilers exploit that traditionally computer memory has been uniformly accessible. A linear function is sufficient to map from an array (multi)index to a memory address, giving efficient access to a memory location. The move towards ultrascale computing is breaking this mapping. Currently we see architectures with collections of manycore processors, connected on fast networks, often with GPUs and other accelerators connected to each processor. The combined memory of such an architecture is no longer linearly addressable, but possibly hierarchical: indexed by processor in the network, then split into core local memory, accelerator local memory, shared core memory and shared accelerator/processor memory. The access time for a memory location varies, depending on where the memory is located, which core/accelerator is accessing the location, and the local, global and collective data access patterns (cache lines, network contention, etc.). On future ultrascale architectures we should expect the data access functions and memory access costs to be more complex.

Programming models to deal with the situation are slow to emerge. The two dominant approaches are explicit processes with message passing (e.g., MPI [15]), and variations of partitioned global address space (PGAS). Both of these models currently assume that memory is distributed across nodes (or cores), but lack support for hierarchical memory and accelerators. The message passing approach is a dramatic change from sequential programming, since a computation here is an ensemble of explicitly communicating programs. Verifying the correctness of concurrent processes is hard. Using a pragmatic *single program multiple data* (SPMD) approach the code transformation to message passing form becomes manageable. The PGAS model is closer to standard programming and is thus easier to reason about. A PGAS compiler may use message passing processes as the target code [8]. Accelerators are mostly supported by specialised models (e.g., Cuda). Hybrid models, e.g., mixing MPI and multicore programming or MPI and accelerators, are being used for hybrid architectures. Porting an application from one programming model to another requires considerable changes to the code. This causes severe challenges to the portability of application between current architectures, and thus may be a severe hindrance for the uptake of efficient future ultrascale architectures.

The emerging gap between problem space arrays and computer memory addressing should be bridged by tools such as the compiler and its support libraries. Keeping tools up to date is a continuous effort as new architectures are being continuously introduced. Unfortunately, tool and compiler vendors are not catching up to the pace of change. For instance, Fortran's take on the PGAS model

was standardised in 2008 [13], almost a decade ago. Yet few Fortran compilers in 2016 support the coarray feature.

This paper promotes the idea that for the hardware space we need a standardised, linear array API encapsulating the heterogeneous memory structure. This can be considered a variation of the PGAS model for distributed memory, and is a refinement of our earlier suggestion [9]. Such an API will empower the software developer to provide their own mapping of the linear indices onto the hierarchical memory structure, in case a relevant one does not already exist. This liberates the developer from relying on compilers and other tools that may never materialise. It also liberates the hardware manufacturer from providing a full fledged tool chain to support every new architecture. A good implementation of the relatively simple API for the new architecture is sufficient. For the problem space we need various adapters mapping, e.g., multiarray or tree structures, to the linear array API. Again, the software developer is empowered to provide relevant mappings in case none exist. Such mappings are obviously reusable across problems with similar needs. The mapping from linear array to hardware structure will be reusable for every application running on that hardware. The mapping from problem space to linear array will be reusable for all applications in the problem space, across all applicable hardware architectures. In order to make such a software architecture to become an efficient tool, it is (1) important to tune the APIs carefully for generic reuse, and (2) to develop applications focussed on using collective operations on the user space abstractions. The former requires careful domain engineering coupled with domain experience. The latter require a change with software developers, who normally are drilled to work with individual elements of arrays and other structures. The proposed approach requires a focus on collective operations on entire data sets. The APIs need to be stable across old and new architectures ensuing portability of application code. The basic linear algebra subprograms (BLAS [12]) is an example of what can be achieved by an API approach. With many of the similar approaches (PGAS, Global Arrays, Coarrays), new notation creeps in to handle new hardware. This causes application portability costs. New notation causes portability issues with existing applications.

The contribution of the paper is to show that an API approach is viable by presenting an API suitable for explicit finite difference solvers. We use collective multiarray operations to develop a solver for Burgers' equation. The multiarray API is mapped to a linear array API. The linear array API is mapped to a plain CPU with linear memory, a GPU local memory, and a GPU local memory with explicit administration of data allocation and deallocation. Since these three distinct hardware mappings all provide the same linear array API, no change is needed in neither the multiarray mapping nor the solver itself. We present the APIs as *concepts* in the sense of [19], i.e., with declarations of types and operations, and axioms describing their properties. Such concepts provide precise description of intended semantics. They work very well with generic implementations (reusable code), and provide verifiable/testable requirements [1].

The paper is organised as follows. In the next section, we introduce the mathematics of our running example, the Burgers' equation, and show how a PDE normally is massaged for implementing a solver. In Section III, we propose our multiarray API, followed by a presentation of the linear array API in Section IV. These two APIs are tied together in Section V. Then we present some experimental results of using our approach to target the Burgers' solver for CPU and GPU implementations. Finally, Section VII discusses some related work before we conclude in Section VIII.

## II. Finite difference numerical software

When writing numerical software, the HPC engineer typically starts from a partial differential equation which is then manipulated into a form suitable for programming. We will use Burgers' equation [3] as an illustration. Burgers' equation is an important nonlinear prototype equation, used for instance in the mathematical modelling of gas dynamics and traffic flow. It is similar to the incompressible Navier-Stokes equation, without the pressure term and external forces like, e.g., gravity. In coordinate free form it reads

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} = \nu \nabla^2 \vec{u}, \tag{1}$$

where $\vec{u}$ denotes velocity, $t$ is time, and $\nu$ is a viscosity coefficient. In one spatial dimension, putting $\vec{u} = (u)$, we get

$$\frac{\partial (u)}{\partial t} + (u) \cdot \nabla (u) = \nu \nabla^2 (u), \tag{2}$$

Choosing Cartesian coordinates, we can elaborate the gradient, the laplacian and the dot product, giving

$$\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2}. \tag{3}$$

We implement the initial value problem $u(x,0) = u_0(x)$, for periodic boundary conditions on an interval of length $L$, $u(x+L,t) = u(x,t)$.

To solve the problem numerically, the standard approach is to discretise the domain. For this we introduce the grid values $u_i^n = u(i\frac{L}{N}, t_n)$ for $i = 0, ..., N-1$, and $t_n = n\Delta t$, where $\Delta t$ is the time step size. In the finite difference method (FDM), we compute a partial derivative by a weighted sum of neighbouring grid points. The weights are formed from two components: (i) a list of factors called the stencil, and (ii) a factor computed from the data resolution (the number of gridpoints). The stencil is carefully decided by a numerical expert. The choice is based on the kind of problem being solved, the initial value being used, accuracy versus speed, etc. For example, in this paper we use the numerical stencils $(-0.5, 0, 0.5)$ and $(1, -2, 1)$ for $\frac{\partial}{\partial x}$ and $\frac{\partial^2}{\partial x^2}$, with factors $\Delta x = \frac{L}{N}$ and $(\Delta x)^2 =$

$\frac{L^2}{N^2}$, respectively. The standard explicit finite difference numerical approximation for equation (3) then becomes,

$$\begin{aligned} u_i^{n+1} &= u_i^n - \frac{\Delta t}{2\Delta x} u_i^n (u_{i+1}^n - u_{i-1}^n) \\ &+ \frac{\nu \Delta t}{(\Delta x)^2} (u_{i+1}^n + u_{i-1}^n - 2u_i^n). \end{aligned} \tag{4}$$

The above approximation is accurate to $\mathcal{O}((\Delta x)^2, \Delta t)$.

The formulation in equation (4) is easy to write up as traditional code: The data for $u^{n+1}$ and $u^n$ is stored in two arrays, one for each, and a single loop, for all element indices 0 through $N-1$, computes $u^{n+1}$ from $u^n$.

Now consider equation (1) in higher spatial dimensions. Using a Cartesian coordinate system, we can write Burgers' equation in three spatial dimensions (3D) as

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} = \nu\frac{\partial^2 u}{\partial x^2} + \nu\frac{\partial^2 u}{\partial y^2} + \nu\frac{\partial^2 u}{\partial z^2} \tag{5}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z} = \nu\frac{\partial^2 v}{\partial x^2} + \nu\frac{\partial^2 v}{\partial y^2} + \nu\frac{\partial^2 v}{\partial z^2} \tag{6}$$

$$\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z} = \nu\frac{\partial^2 w}{\partial x^2} + \nu\frac{\partial^2 w}{\partial y^2} + \nu\frac{\partial^2 w}{\partial z^2} \tag{7}$$

where we have put $\vec{u} = (u, v, w)$. Here we also choose periodic boundary conditions in a 3D domain of length $L_x$, $L_y$ and $L_z$ in the $x$, $y$, and $z$ coordinate directions, respectively. Introducing appropriate grid values, we denote for the first velocity component $u$ that $u^n_{x,i,j,k} = u_x(i\frac{L_x}{N}, j\frac{L_y}{N}, k\frac{L_z}{N}, t_n)$ for $i = 0, ..., N-1$, $j = 0, ..., M-1$ and $k = 0, ..., P-1$. Using analogous approximations as in equation (4), equation (5) can then be discretised as follows.

$$\begin{aligned} u_{i,j,k}^{n+1} &= u_{i,j,k}^n - \frac{\Delta t}{2\Delta x} u_{i,j,k}^n (u_{i+1,j,k}^n - u_{i-1,j,k}^n) \\ &+ \frac{\nu \Delta t}{(\Delta x)^2} (u_{i+1,j,k}^n + u_{i-1,j,k}^n - 2u_{i,j,k}^n) \\ &- \frac{\Delta t}{2\Delta y} v_{i,j,k}^n (u_{i,j+1,k}^n - u_{i,j-1,k}^n) \\ &+ \frac{\nu \Delta t}{(\Delta y)^2} (u_{i,j+1,k}^n + u_{i,j-1,k}^n - 2u_{i,j,k}^n) \\ &- \frac{\Delta t}{2\Delta z} w_{i,j,k}^n (u_{i,j,k+1}^n - u_{i,j,k-1}^n) \\ &+ \frac{\nu \Delta t}{(\Delta z)^2} (u_{i,j,k+1}^n + u_{i,j,k-1}^n - 2u_{i,j,k}^n), \end{aligned} \tag{8}$$

where $\Delta x = \frac{L_x}{N}$, $\Delta y = \frac{L_y}{M}$ and $\Delta z = \frac{L_z}{P}$. Clearly, the two remaining equations (6-7) can be discretised in a similar manner.

Writing traditional style code for the 3D version is more involved than for the 1D case. The data for each of $u, v, w$ is a multiarray with three indices $(i, j, k)$. We will need two sets of multiarrays, one for timestep $n$ and one for the next timestep $n+1$. A triply nested loop is then used to compute timestep $n+1$ from $n$.

```
for (int i = 0; i < N; i++) {
  for (int j = 0; j < M; j++) {
   for (int k = 0; k < P; k++) {
     up(i,j,k) = un(i,j,k)
       - deltat * un(i,j,k)
         * ( un(i+1,j,k) - un(i-1,j,k) )
         / (2 * deltax)
       + ... ;
     vp(i,j,k) = ...;
     wp(i,j,k) = ...;
   }
  }
}
```

Here the suffix `p` is used for variables at timestep $n+1$, the suffix `n` for variables at timestep $n$.

In each elaboration step above, abstractions from the problem domain are unfolded and removed from the exposition. In the end, it is difficult to directly relate the nested loops to the original problem. First, the coordinate-free operators $\nabla$, $\cdot$ and $\nabla^2$ were instantiated with the number of dimensions and Cartesian coordinate system, yielding equation (3) and equations (5-7), for 1 and 3 dimensions, respectively. Then the spatial representation as a finite difference method was chosen, and the continuous operators $\frac{\partial}{dx}$, $\frac{\partial^2}{dx^2}$, ... were instantiated with the corresponding stencils. Thus the forms equation (3) and equations (5-7) bear little resemblance to each other, nor to the problem formulation equation (1). That the resulting code in fact is related to the original problem is non-trivial to validate, and a separate documentation trail needs to be maintained in order to relate the instantiations to the original problem.

Above we have sketched the sequential implementations, almost taking the code directly from the elaborated version of the equations.

## III. A Burgers solver and multiarray API

As initially motivated, we want to reformulate the solver using *collective* operations, i.e., operations that work on the entire array rather than looping through the individual elements. The abstraction level in equation (3) consists of the continuous operators: partial differentiation, addition, multiplication, etc. Considering the FDM discretisation, addition, multiplication, etc, are simple pointwise operations on the array, while partial differentiation relies on neighbouring data.

First we assume an indexing function which returns the element given by the multiindex $(i, j, k)$.

```
1 function get (a:MA, i:int, j:int, k:int) : E;
```

The type `MA` is the multiarray, `int` is an integer type used for indexing data, and `E` is the element type (floating point number).

Next we investigate *mapped* elemental operations, like $+$, $*$, $-$. Mapped functions can be defined as the following concept.

```
/** Elemental addition, multiplication and subtraction. */
  function _+_ ( a:E, b:E ) : E;
  function _*_ ( a:E, b:E ) : E;
  function _-_ ( a:E, b:E ) : E;
  function -_ ( a:E ) : E;
/** Mapped addition, multiplication and subtraction. */
  function _+_ ( a:MA, b:MA ) : MA;
  function _*_ ( a:MA, b:MA ) : MA;
  function _-_ ( a:MA, b:MA ) : MA;
  function -_ ( a:MA ) : MA;

/** Relating the mapped and elemental operations. */
  axiom binaryMap ( a:MA, b:MA, i,j,k:int ) {
    assert get( a+b, i,j,k )
      == get(a,i,j,k) + get(b,i,j,k);
    assert get( a*b, i,j,k )
      == get(a,i,j,k) * get(b,i,j,k);
    assert get( a-b, i,j,k )
      == get(a,i,j,k) - get(b,i,j,k);
  }
  axiom unaryMap ( a:MA, i,j,k:int ) {
    assert get( -a, i,j,k ) == - get(a,i,j,k);
  }
```

The assertions in the axiom must hold for all combinations of input data, the parameters, to the axiom. An axiom is like a procedure, whose intended effect is to validate the assertions on the input data. This can be used to test the correctness of the code, though testing on floating point data seldom works as intended.

To provide the partial difference operators we will need a *shift* function on the multiarrays.

```
  function shift (a:MA, dir:int, d:int) : MA;
```

Here the parameter `dir` instructs which direction we will be shifting (1 for $x$ direction or index i, 2 for $y$ direction or index j, 3 for $z$ direction or index k), and `d` gives the shift distance ($\pm 1$ for one step as needed in the example).

```
  axiom multiarrayShiftAxiom
    ( a:MA, d:int, i,j,k:int ) {
    assert get( shift(a,1,d), i,j,k)
      == get(a, (Lx+i+d)%Lx,j,k);
    assert get( shift(a,2,d), i,j,k)
      == get(a, i,(Ly+j+d)%Ly,k);
    assert get( shift(a,3,d), i,j,k)
      == get(a, i,j,(Lz+k+d)%Lz);
  };
```

Using the modulus operator % for index manipulation above, we define a circular shift, as needed for circular boundary conditions.

With this sketch of the multiarray API, the indexing, map and shift operations, in place, we can for any stencil define the partial derivatives as collective operations on a multiarray. The function `partial1` implements a 1st order partial derivative using a $(-0.5, 0, 0.5)$ stencil, and the function `partial2` implements a 2nd order partial derivative using a $(1, -2, 1)$ stencil.

```
  function partial1 ( a:MA, dir:int ) : MA {
    return (shift(a,dir,1) - shift(a,dir,-1))
      / (2 * deltax );
  };
  function partial2 ( a:MA, dir:int ) : MA {
    return
      (shift(a,dir,-1) - 2*a + shift(a,dir,1))
      / (deltax * deltax );
  };
```

The `dir` argument encodes the direction, `dir==1` for x-direction $\frac{\partial a}{\partial x}$ and $\frac{\partial^2 a}{(\partial x)^2}$, `dir==2` for y-direction $\frac{\partial a}{\partial y}$ and $\frac{\partial^2 a}{(\partial y)^2}$, and `dir==3` for z-direction $\frac{\partial a}{\partial z}$ and $\frac{\partial^2 a}{(\partial z)^2}$.

The solver step for equations (5-7) can now be coded using these operations.

```
  up = un
       + nu*deltat*
       ( partial2(un,1)
       + partial2(un,2)
       + partial2(un,3))
     - deltat*un*partial(un,1)
     - deltat*vn*partial(un,2)
     - deltat*wn*partial(un,3);
  vp = ...;
  wp = ...;
```

Notice how we easily may change the stencil for this computation: it is encapsulated in the partial derivative functions, so replacing these with functions for another stencil is all it takes. The stencil is no longer embedded all over in the formulation of the solver, as it was in equation (4)

## IV. Linear array API for abstracting hardware

Instead of implementing the multiarray API directly in the hardware, we propose a linear API for the mapping onto the hardware. The linear API is slightly more convoluted than the multiarray API, but is often more straight forward to implement on a target hardware architecture.

For this exposition, the linear array needs the following primitive operation to access an element based on an integer index.

```
  /** Get the element at the index position i. */
  function get ( a:A, i:int ) : E;
```

The type `A` is an array of elements and `E` is the element type, typically floating point numbers.

We have a similar mapping of elemental functions for the linear

```
/** Shifts the grp-sized groups of data d positions circularly to the left within each seg-sized segment. */
function shiftSegmentGroups ( a:A, seg:int, grp:int, d:int) : A
  guard seg % grp == 0 && getSize(a) % seg == 0 && abs(d) <= seg;

axiom shiftSegmentGroupsDefinitionAxiom ( a:A, seg:int, grp:int, d:int, j:int ) {
  var size = getSize(a);
  assert size % seg == 0 && seg % grp == 0 && abs(d) <= seg && 0 <= j && j < size;
    // local index within a segment
    var si = j % seg;
    //normalize actual shift value to perform within a segment
    var sh = (grp * (seg + d)) % seg;
    // new index within segment after the local shift
    var ni = ( seg+si-sh ) % seg;
    // obtain the global position of ni within the whole array
    var ind = idiv(j, seg)*seg + ni;
  assert get(shiftSegmentGroups(a,seg,grp,d),ind) == get(a,j);
};
```

*Figure 1: Definition of* `shiftSegmentGroups` *operation for the linear array.*

array as we did for the multiarray.

```
/** Elemental addition, multiplication and subtraction. */
  function _+_ ( a:E, b:E ) : E;
  function _*_ ( a:E, b:E ) : E;
  function _-_ ( a:E, b:E ) : E;
  function -_ ( a:E ) : E;
/** Mapped addition, multiplication and subtraction. */
  function _+_ ( a:A, b:A ) : A;
  function _*_ ( a:A, b:A ) : A;
  function _-_ ( a:A, b:A ) : A;
  function -_ ( a:A ) : A;

/** Relating the mapped and elemental operations. */
axiom binaryMap ( a:A, b:A, i:int ) {
  assert get( a+b, i ) == get(a,i) + get(b,i);
  assert get( a*b, i ) == get(a,i) * get(b,i);
  assert get( a-b, i ) == get(a,i) - get(b,i);
}
axiom unaryMap ( a:A, i:int ) {
  assert get( -a, i ) == - get(a,i);
}
```

We also need to rearrange (permute) the data of the array in various ways for different purposes. Here we provide a fairly general shift operation, see figure 1. It shifts groups of data within segments of the array. The group size must divide the segment size, the segment size must divide the actual array size, and the shift distance must at most be equal to the segment size. (this is written in the guard phrase, which captures the precondition for the shift function). The axiom similarly asserts the relevance of its input data, then nails down the behaviour of this shift function.

These are the linear array operations we need to define and implement for explicit finite difference solvers for PDEs. For other application domains the linear array API may need to contain further operations. Typically a linear API will also provide collective operations like the prefix scan and fold/reduce. These are not covered here.

In section VI.1 we sketch some hardware oriented implementations of this API.

## V. Multiarray library

In section III we defined a multiarray API, and in the previous section we defined a linear array API to mask hardware. Here we explain how to provide a multiarray library on top of the linear array API.

First we define how to retrieve an element from the linear array using a multiindex. This is a bijective, simple linear mapping from a multilinear array with size $L_x$ by $L_y$ by $L_z$ to a linear array of size $L_xL_yL_z$.

```
function get( a:MA, i,j,k:int ) : E {
  return get(a, i*Ly*Lz + j*Lz + k );
}
```

The map functions are straight forward to reuse from the linear array. The maps are pointwise, and thus irrespective of indexing, the result will be at the correct position.

The multiarray shift similarly needs to match both the multiar-

```
assert get( shift(a,1,d), i,j,k) == get(a, (Lx+i+d)%Lx,j,k);
assert get( shiftSegmentGroups(a,Lx*Ly*Lz,Ly*Lz,d), i*Ly*Lz + j*Lz + k )
 == get(a, ((Lx+i+d)%Lx)*Ly*Lz + j*Lz + k );
assert get( shiftSegmentGroups(a,Lx*Ly*Lz,Ly*Lz,d), i*Ly*Lz + j*Lz + k )
 == get( shiftSegmentGroups(a,Lx*Ly*Lz,Ly*Lz,d),
        ((Lx+i+d)%Lx)*Ly*Lz + j*Lz + k + (Lx*Ly*Lz - d*Ly*Lz) % (Lx*Ly*Lz) );
assert get( shiftSegmentGroups(a,Lx*Ly*Lz,Ly*Lz,d), i*Ly*Lz + j*Lz + k )
 == get( shiftSegmentGroups(a,Lx*Ly*Lz,Ly*Lz,d),
        ((Lx+i+d)%Lx)*Ly*Lz + j*Lz + k + ((Lx-d) % Lx)*Ly*Lz );
assert get( shiftSegmentGroups(a,Lx*Ly*Lz,Ly*Lz,d), i*Ly*Lz + j*Lz + k )
 == get( shiftSegmentGroups(a,Lx*Ly*Lz,Ly*Lz,d),
        ((Lx+i+d+ Lx-d)%Lx)*Ly*Lz + j*Lz + k );
```

*Figure 2: Proof for the correctness of the multiarray shift.*

ray's indexing structure and the linear array's shift behaviour, see figure 1. The following defines an appropriate function.

```
function shift ( a:MA, dir:int, d:int ) : MA {
  var seg =
    if dir == 1 then Lx * Ly * Lz
    else if dir == 2 then Ly * Lz
    else /* dir == 3 */       Lz;
    end end;
  var grp = seg /
    if dir == 1 then Ly * Lz
    else if dir == 2 then Lz
    else /* dir == 3 */    1;
    end end;
  return shiftSegmentGroups( a, seg, grp, d );
}
```

We sketch a proof of correctness for the $x$ direction in figure 2. The first assert is from `multiarrayShiftAxiom`. In the next assert we have inserted the multiarray `get` and `shift` algorithms above. The third assert uses `shiftSegmentGroupsDefinitionAxiom` to replace the right hand side with an expression involving `shiftSegmentGroups`. The remaining lines simplify the right hand side until it is clear it matches the left hand side expression. The proof for the $y$ and $z$ directions follow a similar pattern.

The necessary abstractions to code FDM solvers at the continuous level requires a multiarray shift and mapped $+,-,*,/$ on the multiarray. This now boils down to providing `shiftSegmentGroups` and the mapped functions $+,-,*,/$ on an ordinary linear array. This API is quite simple with a few recurring patterns: (i) the map operations, representing a local, per element computation, (ii) the shift operation, representing data reorganisation and communication. Compared to rewriting the entire application code for each architecture, implementing this limited set of functions will be rather trivial—empowering the user to implement hardware specific array libraries if the hardware vendor does not provide it.

## VI. Runtime experiments

We have done several runtime experiments with the developed 3D Burgers' solver. It uses the form from equations (5-7). The runtime experiments target the following two issues.

- Does the suggested approach support easy porting of code between architectures?
  To answer this question we provide implementations of the proposed array API for several architectures, and validate that the application using the API, without source code modification, will run on the relevant hardware.
- Does the application code scale as expected on the various architectures?
  To answer this question we run the application on varying data sizes. For our application example, a 3D FDM Burgers solver, we should see linear scaling with respect to data set size, modulo any effects of caching and virtual memory.

### VI.1 Linear array implementations

Currently we have targeted two hardware architectures for the linear array abstraction.

**CPU C++** A plain sequential implementation for a single CPU and uniform memory. This uses C++ arrays for the linear array API. The mapped operations are each wrapping a loop sequentially performing the lifted operation element by element. The `shiftSegmentGroups` operation makes a temporary copy of the current data, then overwrites the argument array with the shifted data.

**Cuda** An Nvidia GPU version implemented in Cuda. The array is represented as a linear structure in device (GPU) memory, avoiding transfer of data between CPU and GPU memory during the computation. The 5 lifted functions, $+, -, *, /$, are implemented as device loops in Cuda, then called to be executed in multithreading GPU kernel mode. This implies no internal synchronisation, but each mapped operation must be complete before the next operation is started. `shiftSegmentGroups` can be implemented by either synchronising the shifted data between the GPU thread blocks using multiple kernel invocations, or by obtaining the result of the shift operation in a fresh array across the GPU device, eliminating the need for explicit synchronisation within the function. We have chosen the latter, keeping the shift function as a single kernel call. This causes a larger memory use than strictly needed, but is not detrimental to efficiency if the application still fits into GPU memory. If this is not the case, other approaches may be beneficial.

During a computation temporary data is continuously created as subexpressions are evaluated, and subsequently released when the result of the expression is assigned to a variable. On the GPU allocating and deallocating data takes a significant amount of time, so yet another version of the linear array library was created.

**CudaBuffer** An Nvidia GPU version implemented in Cuda as above, but where a buffer large enough to store all temporary device data is created at the start. This is then managed explicitly under the hood by the linear array implementing code, possibly giving more efficient reuse of GPU memory when temporary variables are created and deleted.

This provides an affirmative answer to our first research question: we have achieved portability at the application level by a problem specific API.

These implementations do not attempt any clever optimisations. For instance, map fusion (loop merging) could give significant speedups. This entails rearranging the expressions of the PDE solver, such that local data is only iterated once on the cores, not once per operation. For instance, mapping $A = f(A,B,C)$ for an elemental function $f(a,b,c) = a*b+c$ typically is faster than mapping each of the operations $+, *$ in $A = A*B+C$. Such rewriting should be tool supported, otherwise the clarity, and possibly the portability, of the code will be sacrificed for efficiency.

Even for languages that natively support lifted operations, the efficiency of mapped operations is an important aspect. For instance, in early Fortran 90 compilers, executing $A = A + B$ was much slower for the multiarray version than the corresponding nested loop version.

## VI.2 Runtime results

We configured the software for varying data sizes, each chosen data size doubling the memory requirement for the program. The data, 10 waves of sine functions in the $z$ direction, was generated in the appropriate resolution. Since we are working with 3D data, we double the size of the data set (the size of the linear array), whenever the number of elements in each direction in increased by a factor of $\sqrt[3]{2} \approx 1.26$. We used data set sizes 78MB ($50^3$ elements), 156MB ($60^3$), 307MB ($79^3$), 624MB ($100^3$), 1 248MB ($126^3$), 2 508MB ($159^3$), 4 992MB ($200^3$), and 9 985MB ($252^3$). Each problem size was executed for $1, 10, 100, 1000$ timesteps in each of the three versions (CPU, GPU, GPU buffered), yielding a total of $8*4*3 = 96$ runs. The applications were run on the department's compute server `lyng`. It has Intel Xeon CPU E5-2699 v3 at 2.30GHz cores and Nvidia Tesla K40m with 2880 CUDA Cores at 745 MHz. The runtime is wall clock time. The clock was started immediately before the time iteration of the solver, and stopped immediately after the time iteration. This eliminates unpredictable overhead in starting especially the GPU (Cuda, CudaBuffer) applications. The overhead includes a Cuda just-in-time compilation of GPU code and initialisation of device data, which together may take several seconds even for small datasets. The CPU does not exhibit similar disparity between total execution times and the solver's timestepping loop times.

The CPU timings are tabulated below. The row captions show the linear data sizes, the column captions show the number of iteration timesteps, and the table data is the software's runtime in seconds.

| Cpu C++ | 1 | 10 | 100 | 1000 |
|---|---|---|---|---|
| 50 | 0.268 | 2.900 | 26.198 | 264.116 |
| 63 | 0.535 | 5.794 | 53.620 | 530.304 |
| 79 | 1.058 | 11.516 | 103.989 | 1032.465 |
| 100 | 2.176 | 21.374 | 220.488 | 2181.002 |
| 126 | 4.671 | 51.581 | 482.747 | 4642.034 |
| 159 | 10.156 | 128.887 | 1176.986 | 10124.613 |
| 200 | 28.517 | 958.223 | 6532.079 | 32186.134 |
| 252 | 776.532 | 2387.521 | 13636.333 | 120525.580 |

These runs were concurrent with other loads on the computer, leaving about 10GB of free memory for our application. The results scale well for the smaller tests: it roughly doubles with data set size for the 4 smaller data sizes, and scales linearly with the number of timesteps for the 6 smaller data sizes. The two largest data set sizes behave somewhat erratic, see figure 3, possibly due to swap behaviour when memory ran low.

The Cuda timings are tabulated below. The row captions show the linear data sizes, the column captions show the number of iteration timesteps, and the table data is the software's runtime in seconds.

| Cuda | 1 | 10 | 100 | 1000 |
|---|---|---|---|---|
| 50 | 0.290 | 3.151 | 29.479 | 286.206 |
| 63 | 0.310 | 3.408 | 31.190 | 311.993 |
| 79 | 0.370 | 3.939 | 36.294 | 363.476 |
| 100 | 0.487 | 4.848 | 49.076 | 492.145 |
| 126 | 0.747 | 7.557 | 75.908 | 749.141 |
| 159 | 1.325 | 14.401 | 132.114 | 1327.804 |
| 200 | 3.227 | 31.472 | 310.946 | 3098.445 |
| 252 | 6.448 | 62.974 | 638.149 | 6358.311 |

These runs had exclusive access to the GPU. We see the runtimes grow linearly with the number of timesteps for all data set sizes, see figure 4. For the 4 larger data set sizes the runtime roughly doubles when the data set size doubles. However, the 4 smaller data set sizes do not double in runtime as the data set sizes double. This indicates overheads on the GPU for these smaller data set sizes, possibly related to allocation/deallocation of temporary variables.

The Cuda buffered timings are tabulated below. The row captions show the linear data sizes, the column captions show the number of iteration timesteps, and the table data is the software's runtime in seconds.

| CudaBuffer | 1 | 10 | 100 | 1000 |
|---|---|---|---|---|
| 50 | 0.052 | 0.577 | 5.057 | 50.619 |
| 63 | 0.095 | 1.045 | 9.552 | 94.447 |
| 79 | 0.162 | 1.777 | 16.178 | 162.277 |
| 100 | 0.290 | 2.934 | 29.526 | 292.908 |
| 126 | 0.563 | 5.694 | 57.514 | 567.513 |
| 159 | 1.133 | 12.439 | 114.399 | 1134.132 |
| 200 | 2.346 | 26.430 | 233.351 | 2477.200 |
| 252 | 5.108 | 49.518 | 500.261 | 5032.021 |

These runs show very good scaling in both dimensions, see figure 5. For each data set size, the runtime scales linearly with the number of timesteps, and the runtime roughly doubles when data set size is doubled.

Comparing the two Cuda versions against each other, we notice that for the smaller data set sizes, the unbuffered versions are 5-6 times slower than the buffered version. For the larger data set sizes, this difference is down to 25%. This is still a noticeable speedup for the buffered over the non-buffered GPU version.

Comparing the Cuda version with the CPU version, we see that the CPU does well for the smallest data set size. As the data set sizes grow, the CPU slows significantly down compared to the GPU executions, at times becoming a factor of 20 slower. Comparing the CPU to the buffered Cuda version, we see a slowdown factor close to 30 for some instances. In the figures 3-5, both Cuda versions (the two lower diagrams), are on the same scale, while the CPU version (the upper diagram) has an extra line for 100 000 seconds. This just indicates the common observation that in the PDE domain significant speedups can be achieved using parallel GPU computing over standard single-threaded CPU computing.

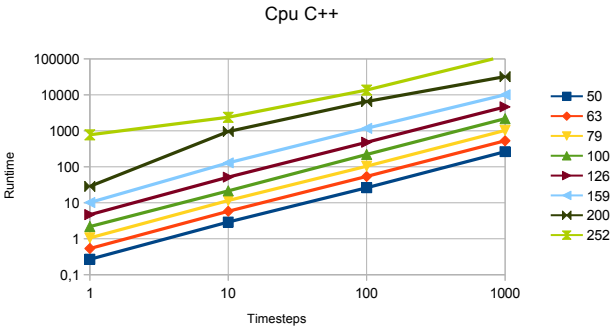


*Figure 3: Runtimes for Burgers 3D solver on main CPU using collective operations implemented in plain C++.*

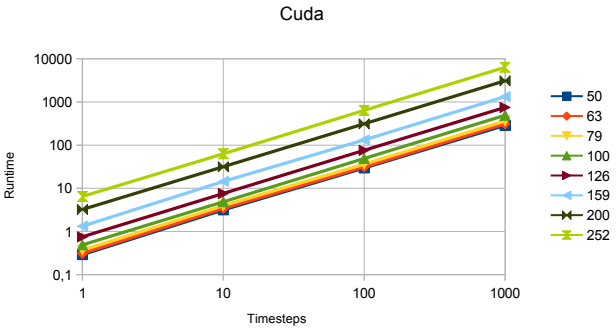


*Figure 4: Runtimes for Burgers 3D solver on GPU using collective operations implemented in Cuda.*

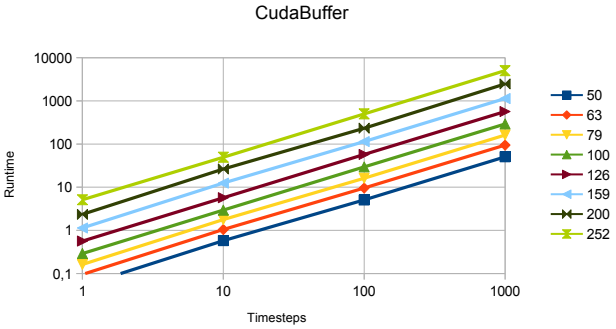


*Figure 5: Runtimes for Burgers 3D solver on GPU using collective operations with explicit buffer management implemented in Cuda.*

We have previously promoted similar ideas in [9] where we demonstrated an implementation using MPI of a multiarray abstraction. Again the data showed similar scaling.

The experiments confirm that the proposed abstraction layered approach to utilising heterogeneous architectures works. At least on the chosen kind of example (explicit finite difference solver), the measured runtimes roughly scale as expected for each type of backend.

## VII. Related Work and Discussion

Abstracting away parts of numerical computations has long been recognised as the path to increase numerical software development productivity and flexibility. As a result, either language extensions or reusable software libraries have been proposed to raise the abstraction level. In the past decade, high-level parallelisation aspects of numerical code has also emerged as an active research field and most proprietary and open source software used in computational science tackle this issue to some extent. We will briefly summarise three approaches: directives, language extensions and libraries.

Common directives-based languages are OpenMP [4] and OpenACC [17]. Directives provide meta-information about the code, enabling the compiler to parallelise and distribute it across cores on a parallel architecture. These are fully dependent on compiler support, and the user cannot adapt the tools to deal with new hardware architectures. Directives are not compatible with our proposed API approach.

Fortran 2008 [13] is a programming language standard with explicit support for parallelism in the form of coarrays. Using coarrays require changing the sequential code, a change that may influence the structure of the entire program [10]. However, a coarray adapted program may also execute on sequential architectures, in principle making the code portable. The coarray feature has been designed for the PGAS model, but provides only rudimentary support for the feature. Work is being done to provide a reusable, open source support for coarrays [8]. The initiative builds on the MPI library, see discussion below. Some authors have proposed extensions to Fortran to handle accellerators [16].

There is a wide range of libraries for supporting parallel and distributed programming. We mention a few here.

C++ [2] has no native support for parallelism, but there are many libraries supporting multiarrays and parallelism (boost.org, Blitz++[20, 7]). It is easy to implement our proposed API structure as a C++ library.

Cuda [5] is an extension to C, C++ and Fortran providing facilities for using Nvidia GPUs. It makes GPGPU programming straightforward, but the code is not portable to other parallel architectures or competing GPU vendors. We use Cuda in our GPU implementation of Burgers' equation.

OpenCL [14] is an extension to C providing interfaces to many different hardware backends, e.g., GPUs and FPGAs. The parallel programming features are low level, but should be well suited for writing the lower level parallel libraries in our proposed API structure.

MPI [15] is a widespread library for explicit communication of data. The library is available for most programming languages, and is adapted to almost all current parallel architectures. Using the library directly is intrusive and forces significant rewrite of source code. It is used as the standard low level communication library, and can easily be used for implementing our low level linear array abstraction.

Diffpack [11] is a proprietary C++ library based on object-oriented numerical code widely used in CSE applications and simulations. Diffpack has become successful due to the powerful abstractions imposed on numerial code offering productivity and efficient code. This provides a domain oriented API as proposed in our approach, but Diffpack does not empower their user to provide their own architecture mappings as we suggest.

Mathworks has an extensive parallel Computing Toolbox for Matlab [6]. These are based on the multiarray abstraction, and provides backends for many parallel architectures. It is possible to implement our proposed API structure in Matlab, but the user is dependent on the vendor for adaption to new architectures.

## VIII. Conclusion

Software structure is very important for the versatility of software, specifically the ability of re-targeting a numerical solver for new HPC architectures. We argue that carefully creating a system of APIs for computational software is a way of organising software achieving this. With object-oriented numerics programming styles becoming embraced also in HPC [18], abstraction oriented approaches are now part of the HPC toolbox.

A well designed library API will embody the application domain's concepts, in such a way that a clean and natural separation occurs between application code and, in our case, the underlying hardware architectures. A message passing library, e.g., MPI, does not have such a property wrt PDE solvers, while an array based library does.

We have proposed using simple array based APIs as a means of abstracting over hardware and providing the applications with a stable abstraction layer. The approach empowers the user to provide their own mappings to heterogenous architectures. Empowering the user to easily re-target a code for new architectures is important to prepare for ultrascale computing.

Compiler vendors seem to a limited extent be able to support this fast changing landscape, hence leaving compiler dependent software support in the dark. Many language extensions for parallelism also fail in portability, requiring more or less intrusive rewrites of code when porting between architectures.

The technical results show that our approach is feasible and delivers on two important issues: (I) the approach makes applications portable across varying hardware architectures without modifications in the application source code, and (II) the approach achieves the expected runtime scalability to be useful for HPC. We have thus converted a portability problem into a much simpler library implementation problem.

Future work includes building further benchmarks for more complex hardware architectures, and comparing our results to those

achieved by the more labour intensive standard approaches. Further we want to expand the ideas to other problem domains.

## Acknowledgment

This research has in part been financed by The Research Council of Norway through the project Design of a Mouldable Programming Language (DMPL), and has received support from EU under the COST Program Action IC1305, Network for Sustainable Ultrascale Computing (NESUS).

## References

[1] Anya Helene Bagge, Valentin David, and Magne Haveraaen. Testing with axioms in C++ 2011. *Journal of Object Technology*, 10:10:1–32, 2011.

[2] Pete Becker et al. ISO/IEC 14882:2011: Programming languages – C++ (final draft international standard). Technical Report N3290, JTC1/SC22/WG21 – The C++ Standards Committee, April 2011.

[3] J.M. Burgers. A mathematical model illustrating the theory of turbulence. In Richard Von Mises and Theodore Von Kármán, editors, *Advances in Applied Mechanics*, volume 1, pages 171 – 199. Elsevier, 1948.

[4] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[5] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[6] Timothy A. Davis. *MATLAB Primer, Eighth Edition*. CRC Press, Inc., Boca Raton, FL, USA, 8th edition, 2010.

[7] Denis Demidov, Karsten Ahnert, Karl Rupp, and Peter Gottschling. Programming CUDA and opencl: A case study using modern C++ libraries. *SIAM J. Scientific Computing*, 35(5), 2013.

[8] Alessandro Fanfarillo, Tobias Burnus, Valeria Cardellini, Salvatore Filippone, Dan Nagle, and Damian W. I. Rouson. OpenCoarrays: Open-source transport layers supporting coarray Fortran compilers. In Allen D. Malony and Jeff R. Hammond, editors, *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, PGAS 2014, Eugene, OR, USA, October 6-10, 2014*, pages 4:1–4:11. ACM, 2014.

[9] Magne Haveraaen. Machine and collection abstractions for user-implemented data-parallel programming. *Scientific Programming*, 8(4):231–246, 2000.

[10] Magne Haveraaen, Karla Morris, Damian W. I. Rouson, Hari Radhakrishnan, and Clayton Carson. High-performance design patterns for modern Fortran. *Scientific Programming*, 2015:942059:1–942059:14, 2015.

[11] Hans Petter Langtangen. *Computational Partial Differential Equations - Numerical Methods and Diffpack Programming*, volume 1 of *Texts in Computational Science and Engineering*. Springer, 2003.

[12] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979.

[13] M. Metcalf, J. Reid, and M. Cohen. *Modern Fortran Explained*. Oxford University Press, 2011.

[14] Aaftab Munshi, Benedict Gaster, Timothy G. Mattson, James Fung, and Dan Ginsburg. *OpenCL Programming Guide*. Addison-Wesley Professional, 1st edition, 2011.

[15] Peter S Pacheco. *Parallel programming with MPI*. Morgan Kaufmann, 1997.

[16] Craig Rasmussen, Matthew Sottile, Soren Rasmussen, Dan Nagle, and William Dumas. Cafe: Coarray Fortran extensions for heterogeneous computing. In *Proceedings IPDPS Workshops*, page to appear, 2016.

[17] Ruymán Reyes, Iván López, Juan J. Fumero, and Francisco Sande. A preliminary evaluation of openacc implementations. *J. Supercomput.*, 65(3):1063–1075, September 2013.

[18] Damian W.I. Rouson, Jim Xia, and Xiaofeng Xu. *Scientific Software Design: The Object-Oriented Way*. Cambridge University Press, 2011.

[19] Alexander Stepanov and Paul McJones. *Elements of Programming*. Addison-Wesley Professional, 1st edition, 2009.

[20] Todd L. Veldhuizen. Blitz++: The library that thinks it is a compiler. In Hans Petter Langtangen, Are Magnus Bruaset, and Ewald Quak, editors, *Advances in Software Tools for Scientific Computing*, volume 10 of *Lecture Notes in Computational Science and Engineering*, pages 57–87. Springer Berlin Heidelberg, 2000.